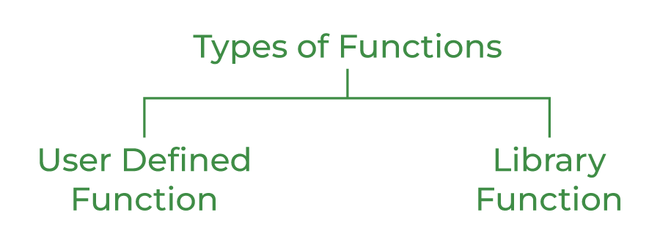
**Functions**

* A function is a block of code which only runs when it is called.
* You can pass data, known as parameters, into a function.
* Functions are used to perform certain actions, and they are important for reusing code: Define the code once, and use it many times.



### **Library Function**

Library functions are also called “***builtin Functions***“. These functions are a part of a compiler package that is already defined and consists of a special function with special and different meanings. Builtin Function gives us an edge as we can directly use them without defining them whereas in the user-defined function we have to declare and define a function before using them.

**For Example:** sqrt(), setw(), strcat(), etc.

### 

### 

### **User Defined Function**

User Defined functions are user/customer-defined blocks of code specially customized to reduce the complexity of big programs. They are also commonly known as “***tailor-made functions***” which are built only to satisfy the condition in which the user is facing issues meanwhile reducing the complexity of the whole program.

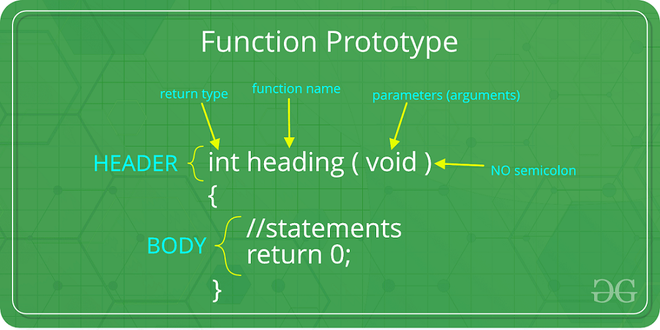
**ADVANTAGES**

* Functions help us in ***reducing code redundancy***. If functionality is performed at multiple places in software, then rather than writing the same code, again and again, we create a function and call it everywhere. This also helps in maintenance as we have to change at one place if we make future changes to the functionality.
* Functions make code ***modular***. Consider a big file having many lines of code. It becomes really simple to read and use the code if the code is divided into functions.
* Functions provide ***abstraction***. For example, we can use library functions without worrying about their internal work.

## Function Declaration and Definition

A C++ function consist of two parts:

* Declaration: the return type, the name of the function, and parameters (if any)
* Definition: the body of the function (code to be executed)



## Create a Function

C++ provides some pre-defined functions, such as main(), which is used to execute code. But you can also create your own functions to perform certain actions.

To create (often referred to as *declare*) a function, specify the name of the function, followed by parentheses ():

### Syntax

void *myFunction*() {

// code to be executed

}

#### Example Explained

* myFunction() is the name of the function
* void means that the function does not have a return value. You will learn more about return values later in the next chapter
* inside the function (the body), add code that defines what the function should do

## Call a Function

Declared functions are not executed immediately. They are "saved for later use", and will be executed later, when they are called.

To call a function, write the function's name followed by two parentheses () and a semicolon ;

In the following example, myFunction() is used to print a text (the action), when it is called:

### Example

Inside main, call myFunction():

// Create a function

void myFunction() {

cout << "I just got executed!";

}

int main() {

myFunction(); // call the function

return 0;

}

// Outputs "I just got executed!"

A function can be called multiple times:

### Example

void myFunction() {

cout << "I just got executed!\n";

}

int main() {

myFunction();

myFunction();

myFunction();

return 0;

}

// I just got executed!

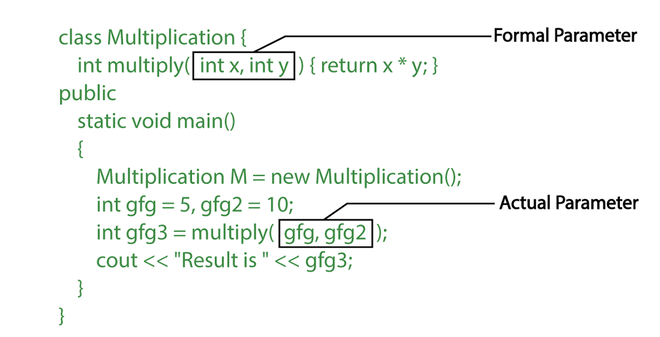
// I just got executed!

// I just got executed!

## **Parameter Passing to Functions**

The parameters passed to function are called ***actual parameters***. For example, in the program below, 5 and 10 are actual parameters.

The parameters received by the function are called ***formal parameters***. For example, in the above program x and y are formal parameters.



**There are two most popular ways to pass parameters:**

1. ***Pass by Value:*** In this parameter passing method, values of actual parameters are copied to the function’s formal parameters and the two types of parameters are stored in different memory locations. So any changes made inside functions are not reflected in the actual parameters of the caller.
2. ***Pass by Reference:*** Both actual and formal parameters refer to the same locations, so any changes made inside the function are actually reflected in the actual parameters of the caller.

## **Difference between call by value and call by reference in C++**

| **Call by value** | **Call by reference** |
| --- | --- |
| A copy of value is passed to the function | An address of value is passed to the function |
| Changes made inside the function is not reflected on other functions | Changes made inside the function is reflected outside the function also |
| Actual and formal arguments will be created in different memory location | Actual and formal arguments will be created in same memory location |

# **Function Overloading in C++**

Function overloading is a feature of object-oriented programming where two or more functions can have the same name but different parameters. When a function name is overloaded with different jobs it is called Function Overloading. In Function Overloading “Function” name should be the same and the arguments should be different. Function overloading can be considered as an example of a [polymorphism](https://www.geeksforgeeks.org/polymorphism-in-c/) feature in C++.

The parameters should follow any one or more than one of the following conditions for Function overloading:

* Parameters should have a different type

*add(int a, int b)*

*add(double a, double b)*

#include <iostream>

using namespace std;

void add(int a, int b)

{

cout << "sum = " << (a + b);

}

void add(double a, double b)

{

cout << endl << "sum = " << (a + b);

}

// Driver code

int main()

{

add(10, 2);

add(5.3, 6.2);

return 0;

}

**Output**

sum = 12

sum = 11.5

## **How does Function Overloading work?**

* *Exact match*:- (Function name and Parameter)
* *If* a *not exact match is found:*–

->Char, Unsigned char, and short are promoted to an int.

->Float is promoted to double

* *If no match is found*:

->C++ tries to find a match through the standard conversion.

* *ELSE ERROR*

# **Inline Functions in C++**

Inline function is one of the important feature of C++. So, let’s first understand why inline functions are used and what is the purpose of inline function?

When the program executes the function call instruction the CPU stores the memory address of the instruction following the function call, copies the arguments of the function on the stack and finally transfers control to the specified function. The CPU then executes the function code, stores the function return value in a predefined memory location/register and returns control to the calling function. This can become overhead if the execution time of function is less than the switching time from the caller function to called function (callee). For functions that are large and/or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run. However, for small, commonly-used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function’s code. This overhead occurs for small functions because execution time of small function is less than the switching time.

C++ provides an inline functions to reduce the function call overhead. Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. This substitution is performed by the C++ compiler at compile time. Inline function may increase efficiency if it is small.

The syntax for defining the function inline is:

inline return-type function-name(parameters)

{

// function code

}

Remember, inlining is only a request to the compiler, not a command. Compiler can ignore the request for inlining. Compiler may not perform inlining in such circumstances like:

1) If a function contains a loop. (for, while, do-while)

2) If a function contains static variables.

3) If a function is recursive.

4) If a function return type is other than void, and the return statement doesn’t exist in function body.

5) If a function contains switch or goto statement.

**Inline functions provide following advantages:**

1) Function call overhead doesn’t occur.

2) It also saves the overhead of push/pop variables on the stack when function is called.

3) It also saves overhead of a return call from a function.

4) When you inline a function, you may enable compiler to perform context specific optimization on the body of function. Such optimizations are not possible for normal function calls. Other optimizations can be obtained by considering the flows of calling context and the called context.

5) Inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function call preamble and return.

**Inline function disadvantages:**

1) The added variables from the inlined function consumes additional registers, After in-lining function if variables number which are going to use register increases than they may create overhead on register variable resource utilization. This means that when inline function body is substituted at the point of function call, total number of variables used by the function also gets inserted. So the number of register going to be used for the variables will also get increased. So if after function inlining variable numbers increase drastically then it would surely cause an overhead on register utilization.

2) If you use too many inline functions then the size of the binary executable file will be large, because of the duplication of same code.

3) Too much inlining can also reduce your instruction cache hit rate, thus reducing the speed of instruction fetch from that of cache memory to that of primary memory.

4) Inline function may increase compile time overhead if someone changes the code inside the inline function then all the calling location has to be recompiled because compiler would require to replace all the code once again to reflect the changes, otherwise it will continue with old functionality.

5) Inline functions may not be useful for many embedded systems. Because in embedded systems code size is more important than speed.

6) Inline functions might cause thrashing because inlining might increase size of the binary executable file. Thrashing in memory causes performance of computer to degrade.

#include <iostream>

using namespace std;

inline int cube(int s)

{

return s\*s\*s;

}

int main()

{

cout << "The cube of 3 is: " << cube(3) << "\n";

return 0;

}

//Output: The cube of 3 is: 27

# **Macros C++**

A [**macro**](https://www.geeksforgeeks.org/c-language-2-gq/macro-preprocessor-gq/) is a piece of code in a program that is replaced by the value of the macro. Macro is defined by **#define** directive. Whenever a macro name is encountered by the compiler, it replaces the name with the definition of the macro. Macro definitions need not be terminated by a semi-colon(**;**).

#include <iostream>

using namespace std;

// Macro definition

#define LIMIT 5

// Driver Code

int main()

{

// Print the value of macro defined

cout << "The value of LIMIT"

<< " is " << LIMIT;

return 0;

}

**Output**

The value of LIMIT is 5

**What is wrong with macro?**

Readers familiar with the C language knows that C language uses macro. The preprocessor replace all macro calls directly within the macro code. It is recommended to always use inline function instead of macro. According to Dr. Bjarne Stroustrup the creator of C++ that macros are almost never necessary in C++ and they are error prone. There are some problems with the use of macros in C++. Macro cannot access private members of class. Macros looks like function call but they are actually not.

Example:

| #include <iostream>  using namespace std;  class S  {  int m;  public:  #define MAC(S::m) // error  }; |
| --- |

# **Difference between Inline and Macro in C++**

| S.NO | Inline | Macro |
| --- | --- | --- |
| 1. | An inline function is defined by the **inline** keyword. | Whereas the macros are defined by the **#define** keyword. |
| 2. | Through inline function, the class’s data members can be accessed. | Whereas macro can’t access the class’s data members. |
| 3. | In the case of inline function, the program can be easily debugged. | Whereas in the case of macros, the program can’t be easily debugged. |
| 4. | In the case of inline, the arguments are evaluated only once. | Whereas in the case of macro, the arguments are evaluated every time whenever macro is used in the program. |
| 5. | In C++, inline may be defined either inside the class or outside the class. | Whereas the macro is all the time defined at the beginning of the program. |
| 6. | In C++, inside the class, the short length functions are automatically made the inline functions. | While the macro is specifically defined. |
| 7. | Inline is not as widely used as macros. | While the macro is widely used. |
| 8. | Inline is not used in competitive programming. | While the macro is very much used in competitive programming. |
| 9. | Inline function is terminated by the curly brace at the end. | While the macro is not terminated by any symbol, it is terminated by a new line. |

**Recursive Functions**

Program to find factorial of a number using recursion.